John von Neumann Institute for Computing

Efficient representation and parallel computation of string-substring longest common subsequences

A. Tiskin

published in

Parallel Computing:
Current & Future Issues of High-End Computing,
Proceedings of the International Conference ParCo 2005,
G.R. Joubert, W.E. Nagel, F.J. Peters, O. Plata, P. Tirado, E. Zapata
(Editors),
John von Neumann Institute for Computing, Jülich,
NIC Series, Vol. **33**, ISBN 3-00-017352-8, pp. 827-834, 2006.

http://www.fz-juelich.de/nic-series/volume33

Efficient representation and parallel computation of string-substring longest common subsequences

Alexander Tiskin[a]

[a]Department of Computer Science, The University of Warwick, Coventry CV4 7AL, UK

Given two strings a, b of length m, n respectively, the string-substring longest common subsequence (SS-LCS) problem consists in computing the length of the longest common subsequence of a and every substring of b. An explicit representation of the output lengths is of size $\Theta(n^2)$. We show that the output can be represented implicitly by a set of n two-dimensional integer points, where individual output lengths are obtained by dominance counting queries. This leads to a data structure of size $O(n)$, which allows to query an individual output length in time $O(\frac{\log n}{\log\log n})$, using a recent result by JaJa, Mortensen and Shi. The currently best sequential SS-LCS algorithm by Alves et al. can be adapted to produce the output in the above geometric representation. We also develop a new parallel SS-LCS algorithm that runs on a p-processor coarse-grained computer in $O(\frac{mn}{p})$ local computation, $O(n \log p)$ communication, $O(\log p)$ barrier synchronisations, and $O(n)$ memory per processor, producing the output in the above geometric representation. Compared to previously known results, our approach presents a substantial improvement in algorithm functionality, output representation efficiency, communication efficiency and/or memory efficiency.

1. Introduction

In this paper, we consider the string-substring longest common subsequence (SS-LCS) problem, an important special case of the local sequence alignment problem, which has numerous applications in computational biology (see e.g. [7, Chapter 6], as well as references in [2,3]). Given two strings a, b of lengths m, n respectively, the SS-LCS problem consists in computing the length of the longest common subsequence of a and every substring of b. If the output lengths are represented explicitly, the total size of the output is $\Theta(n^2)$. To reduce the storage requirements, we allow the output lengths to be represented implicitly by a smaller data structure that allows efficient retrieval of individual output values. It is well-known [8,2,3] that a solution to the SS-LCS problem can be represented by a data structure of size $O(n)$. Retrieval of an individual output length typically requires scanning of at least a constant fraction of this data structure, and therefore takes time $O(n)$. In this paper, we show that the output lengths can be represented by a set of n two-dimensional integer points, where individual output lengths are obtained by dominance counting queries. This leads to a data structure of size $O(n)$, that allows to query an individual output length in time $O(\frac{\log n}{\log\log n})$, using a recent result by JaJa, Mortensen and Shi [6]. The described approach presents a substantial improvement in SS-LCS query efficiency over previous approaches.

Alves et al. [3] proposed a sequential SS-LCS algorithm, based on an idea of Schmidt [9], that runs in $O(mn)$ time and $O(n)$ memory, obtaining an implicit representation of the output. This algorithm can be adapted to produce the output in our more efficient geometric representation, without any increase in asymptotic time or memory requirements.

The SS-LCS problem can be solved in the more general setting of computing all boundary-to-boundary longest (or shortest) paths in a weighted grid graph. The first coarse-grained parallel

algorithm for this more general problem was proposed by Alves et al. [1]. The algorithm runs on a p-processor coarse-grained computer in $O(\frac{n^2 \log m}{p})$ local computation, $O(\frac{nm \log p}{p})$ communication, $O(\log p)$ barrier synchronisations, and $O(\frac{nm}{p})$ memory per processor, obtaining the output path lengths explicitly. For the SS-LCS problem proper, this was improved upon by Alves et al. [2], based on ideas of Lu and Lin [8]. Their algorithm runs in $O(\frac{mn}{p})$ local computation, $O(m^{1/2} n \log p)$ communication, $O(\log p)$ barrier synchronisations, and $O(m^{1/2} n)$ memory per processor, obtaining an implicit representation of the output. In this paper, we propose a parallel algorithm that runs in $O(\frac{mn}{p})$ local computation, $O(n \log p)$ communication, $O(\log p)$ barrier synchronisations, and $O(n)$ memory per processor, producing the output in our geometric representation. This is a substantial improvement over [2] in communication and memory efficiency, as well as the output query efficiency.

2. Problem statement and notation

Given a string, we distinguish between its contiguous *substrings*, and not necessarily contiguous *subsequences*. For two strings $a = a_1 a_2 \ldots a_m$ and $b = b_1 b_2 \ldots b_n$ of lengths m, n respectively, the *string-substring longest common subsequence (SS-LCS) problem* consists in computing the length of the longest common subsequence of a and every substring of b.

In addition to standard (non-negative) integer indices $0, 1, 2, \ldots$, we use (non-negative) *odd half-integer*[1] indices $\frac{1}{2}, \frac{3}{2}, \frac{5}{2}, \ldots$. For two integers i, j, we write $i \trianglelefteq j$ if $j - i \in \{0, 1\}$, and $i \triangleleft j$ if $j - i = 1$. We denote

$$[i : j] = \{i, i+1, \ldots, j-1, j\} \qquad \langle i : j \rangle = \left\{i + \tfrac{1}{2}, i + \tfrac{3}{2}, \ldots, j - \tfrac{3}{2}, j - \tfrac{1}{2}\right\}$$

For a function f and a predicate P defined on a variable i, we write $\mathrm{any}_{i:P(i)}\, f(i)$ to denote an arbitrary element of the set $\{f(i) : P(i)\}$. This is analogous to the use of $\min_{i:P(i)} f(i)$ to denote the minimum element of this set[2].

3. Problem analysis

It is well-known that an instance of the SS-LCS problem can be represented by a weighted *grid dag (directed acyclic graph)*, defined on a set of nodes $v_{l,i}$, where $l \in [0 : m]$, $i \in [0 : n]$. The edges are defined as follows:

edge $v_{l,i-1} \to v_{l,i}$ of weight 0, for all $l \in [0 : m]$, $i \in [1 : n]$;
edge $v_{l-1,i} \to v_{l,i}$ of weight 0, for all $l \in [1 : m]$, $i \in [0 : n]$;
edge $v_{l-1,i-1} \to v_{l,i}$ of weight 1 if $a_l = b_i$, for all $l \in [1 : m]$, $i \in [1 : n]$.

A common subsequence of string a and substring $b_{i+1} \ldots b_j$ of length r corresponds to a path $v_{0,i} \rightsquigarrow v_{m,j}$ of total weight r. The solution to the SS-LCS problem is equivalent to finding the weight $A(i, j)$ of a longest (heaviest) path $v_{0,i} \rightsquigarrow v_{m,j}$ for all $i, j \in [0 : n]$. If $i = j$, we have $A(i, j) = 0$. By convention, if $j < i$, then we let $A(i, j) = j - i$.

[1] It would be possible to reformulate all our results using only integers. However, using half-integers helps to make the exposition simpler and more elegant.

[2] In fact, "min" (or "max") can always be used instead of "any"; however, such usage would be somewhat misleading when "any" happens to be sufficient.

Theorem 1. *Values $A(i,j)$ have the following properties:*

$$A(i,j) \trianglelefteq A(i-1,j); \tag{1}$$
$$A(i,j) \trianglelefteq A(i,j+1); \tag{2}$$
$$\text{if } A(i,j+1) \triangleleft A(i-1,j+1), \text{ then } A(i,j) \triangleleft A(i-1,j); \tag{3}$$
$$\text{if } A(i-1,j) \triangleleft A(i-1,j+1), \text{ then } A(i,j) \triangleleft A(i,j+1). \tag{4}$$

Proof. A path $v_{0,i-1} \rightsquigarrow v_{m,j}$ can be obtained by $v_{0,i-1} \rightarrow v_{0,i} \rightsquigarrow v_{m,j}$. Therefore, $A(i,j) \leq A(i-1,j)$. On the other hand, any path $v_{0,i-1} \rightsquigarrow v_{m,j}$ consists of a subpath $v_{0,i-1} \rightsquigarrow v_{l,i}$ of weight at most 1, followed by a subpath $v_{l,i} \rightsquigarrow v_{m,j}$. Therefore, $A(i,j) \geq A(i-1,j) - 1$. We thus have (1) and, by symmetry, (2).

A crossing pair of paths $v_{0,i} \rightsquigarrow v_{m,j}$ and $v_{0,i-1} \rightsquigarrow v_{m,j+1}$ can be rearranged into a non-crossing pair of paths $v_{0,i-1} \rightsquigarrow v_{m,j}$ and $v_{0,i} \rightsquigarrow v_{m,j+1}$. Therefore, we have *the Monge property*:

$$A(i,j) + A(i-1,j+1) \leq A(i-1,j) + A(i,j+1)$$

Rearranging the terms

$$A(i-1,j+1) - A(i,j+1) \leq A(i-1,j) - A(i,j)$$

and applying (1), we obtain (3) and, by symmetry, (4). ■

The properties of Theorem 1 are symmetric with respect to i and $n-j$. Alves et al. [2,3] introduce the same properties but do not make the most of their symmetry. We aim to exploit symmetry to the full.

Corollary 1. *Values $A(i,j)$ have the following properties:*

$$\text{if } A(i,j) \triangleleft A(i-1,j), \text{ then } A(i,j') \triangleleft A(i-1,j') \text{ for all } j' \leq j;$$
$$\text{if } A(i,j) = A(i-1,j), \text{ then } A(i,j') = A(i-1,j') \text{ for all } j' \geq j.$$

Also,

$$\text{if } A(i,j) \triangleleft A(i,j+1), \text{ then } A(i',j) \triangleleft A(i',j+1) \text{ for all } i' \geq i;$$
$$\text{if } A(i,j) = A(i,j+1), \text{ then } A(i',j) = A(i',j+1) \text{ for all } i' \leq i.$$

Proof. In both pairs, the properties are each other's converse and an immediate consequence of Theorem 1. ■

Informally, Corollary 1 says that, if values A are represented as a matrix, then the inequality between the corresponding elements in two successive rows (respectively, columns) "propagates to the left (respectively, downwards)", and the equality "propagates to the right (respectively, upwards)". Recall that by convention, $A(i,j) = j - i$ for all index pairs $j < i$. Therefore, we always have an inequality between the corresponding elements in successive rows or columns in the lower triangular part of matrix A. If we fix i and scan the set of indices j from left ($j = 0$) to right ($j = n$), an inequality may change to an equality at most once. We call such a value of j *critical* for i. Symmetrically, if we fix j and scan the set of indices i from bottom ($i = n$) to top ($i = 0$), an inequality may change to an equality at most once, and we can identify values of i that are critical for j. Crucially, for all pairs (i,j), index i will be critical for j if and only if index j is critical for i. This property lies at the core of our method, which is based on the following definition.

Definition 1. *An odd half-integer point* $(i,j) \in \langle 0:n\rangle^2$ *is called* A-critical, *if*

$$A\big(i+\tfrac{1}{2}, j-\tfrac{1}{2}\big) \lhd A\big(i-\tfrac{1}{2}, j-\tfrac{1}{2}\big) = A\big(i+\tfrac{1}{2}, j+\tfrac{1}{2}\big) = A\big(i-\tfrac{1}{2}, j+\tfrac{1}{2}\big)$$

In particular, point (i,j) is never A-critical for $i > j$. Point (i,j) is A-critical for $i = j$, iff $A\big(i-\frac{1}{2}, j+\frac{1}{2}\big) = 0$.

Corollary 2. *For each* i *(respectively,* j*), there exists at most one* j *(respectively,* i*) such that the point* $(i,j) \in \langle 0:n\rangle^2$ *is* A*-critical.*

Proof. By Corollary 1 and Definition 1. ■

Definition 1 and Corollary 2 allow us to represent an $(n+1)\times(n+1)$ SS-LCS matrix by its set of critical points, which are at most n. The following theorem shows that such representation is unique, and gives a simple formula for recovering matrix elements.

Definition 2. *Point* (i_0,j_0) dominates[3] *point* (i,j), *if* $i_0 < i$ *and* $j < j_0$.

Informally, the dominated point is "below and to the left" of the dominating point.

Theorem 2. *For an arbitrary integer point* $(i_0,j_0) \in [0:n]^2$, *let* $d_A(i_0,j_0)$ *denote the number of* A*-critical points it dominates. We have*

$$A(i_0,j_0) = j_0 - i_0 - d_A(i_0,j_0)$$

Proof. Induction on $j_0 - i_0$. ■

There is a close connection between Theorem 2 and the canonical representation of general Monge matrices (see e.g. [5]). The difference is that in the special case of SS-LCS matrices, the representation size is just $O(n)$.

Informally, Theorem 2 says that the value $A(i_0,j_0)$ is determined by the number of A-critical points dominated by (i_0,j_0). Trivially, this number can be obtained by scanning the set of all critical points in time $O(n)$. Much more efficient methods exist when preprocessing of the critical point set is allowed.

The dominance relationship between two-dimensional (in general, multi-dimensional) points is a classical topic in computation geometry. The following theorems are derived from two relevant geometric results, one classical and one recent.

Theorem 3. *Given an instance of the SS-LCS problem, there exists a data structure which*
- *has size* $O(n \log n)$*;*
- *can be built from the set of critical points in time* $O(n \log n)$*;*
- *allows to query an individual output length in time* $O(\log n)$.

Proof. Use a 2D range tree [4]. The value $A(i_0,j_0)$ is then obtained by Theorem 2. ■

Theorem 4. *Given an instance of the SS-LCS problem, there exists a data structure which*
- *has size* $O(n)$*;*
- *allows to query an individual output length in time* $O(\frac{\log n}{\log\log n})$.

Proof. As in Theorem 3, but the 2D range tree is replaced by the data structure from [6]. ■

While the data structure of Theorem 4 is asymptotically more efficient, the structure of Theorem 3 is simpler, requires a less powerful computation model, and is more likely to be practical.

[3]The standard definition of dominance requires $i < i_0$ instead of $i_0 < i$. Our definition is more convenient in the context of the LCS problem.

4. The parallel algorithm

A divide-and-conquer framework for the SS-LCS problem and the more general string editing problem has been developed in [1–3]. String a is partitioned into substrings, inducing a partitioning of the representation dag into strips, such that each pair of adjacent strips shares a single row of nodes. Consider a pair of adjacent strips defined by nodes $v_{l,i}$, where $l_0 \le l \le l_1$ and $l_1 \le l \le l_2$, respectively. Denote the SS-LCS matrices in the substrips by A, B respectively. The goal is to merge these matrices to obtain the SS-LCS matrix C for the combined strip defined by nodes $v_{l,i}$, where $l_0 \le l \le l_2$.

By Theorem 2, matrices A, B, C can be represented by the sets of at most n A-, B- and C-critical points. For the special case when either of the strips is of width 1 (e.g. $l_2 - l_1 = 1$), Alves et al. [3] describe a merging procedure that runs in time $O(n)$. Their sequential algorithm based on this procedure runs in time $O(mn)$, and produces a data structure of size $O(n)$, equivalent to the set of critical points representing the solution of the original SS-LCS problem. By adding a post-processing phase based on Theorems 3, 4, the algorithm can be adapted to produce a query-efficient representation of the output.

We now present an efficient coarse-grained parallel algorithm for the SS-LCS problem. We assume the *bulk-synchronous parallel (BSP)* computation model. A *BSP computer*, introduced in [10], consists of p *processors* connected by a communication network. Each processor has a fast *local memory*. A BSP computation consists of a sequence of S *supersteps*. with costs $w_s + h_s \cdot g + l$, $1 \le s \le S$, where w_s is the superstep's local computation cost, h_s is the superstep's communication cost, and g, l are parameters of the computer. The overall cost of a BSP computation is $W + H \cdot g + S \cdot l$, where $W = \sum_{s=1}^{S} w_s$ is the total local computation cost, $H = \sum_{s=1}^{S} h_s$ is the total communication cost, and S is the total synchronisation cost.

Our parallel algorithm is based on a novel sequential strip merging procedure, which works for arbitrary strip widths $l_1 - l_0$, $l_2 - l_1$, and runs in time $O(n^{3/2})$. At the base of the recursion, we use the sequential algorithm by Alves et al. [3].

Algorithm 1. *String-substring longest common subsequences in BSP.*

Parameters: integers m, n. We assume $m \ge n^{1/2} p \log p$.

Input: strings a, b of length m and n, respectively.

Output: the set of critical points for strings a, b.

Description. The computation proceeds in two stages.

First stage. Partition string a into p substrings of length m/p, inducing a partitioning of the representation dag into strips, such that each pair of adjacent strips shares a single row of nodes. The resulting p subproblems are distributed across the processors. Each processor runs the algorithm by Alves et al. [3] on the local subproblem, obtaining a critical point set representation of the resulting SS-LCS matrix.

Second stage. Perform $\log p$ levels of pairwise subproblem merging. Each subproblem is represented by a set of at most n critical points. For two subproblems with SS-LCS matrices A, B, the sets of A- and B-critical points are collected, and the set of C-critical points is computed sequentially by a designated processor. The computation proceeds as follows.

By Theorem 2, computing C-critical points is equivalent to determining the set of values

$$d_C(i,k) = \min_{j \in [0:n]} [d_A(i,j) + d_B(j,k)]$$

for $i, k \in [0 : n]$. Assume for simplicity that n is a power of 2. We proceed by partitioning the index set $\langle 0 : n\rangle^2$ recursively into regular half-sized square blocks. For each block, we establish the number of C-critical points contained in it, and proceed with the recursive partitioning of the block as long as this number is greater than 0.

Consider an $h \times h$ block

$$\langle i_0 - h : i_0\rangle \times \langle k_0 : k_0 + h\rangle$$

The C-critical points in this block will be determined by A-critical points in $\langle i_0-h : i_0\rangle\times\langle 0 : n\rangle$, and B-critical points in $\langle 0 : n\rangle \times \langle k_0 : k_0 + h\rangle$. We call such A- and B-critical points *relevant*. For the current block, there are at most h relevant points in each of A, B.

For any $j \in [0 : n]$, let $\delta_A(j)$ (respectively, $\delta_B(j)$) denote the number of relevant A-critical (respectively, B-critical) points in $\langle i_0 - h : i_0\rangle \times \langle 0 : j\rangle$ (respectively, $\langle j : n\rangle \times \langle k_0 : k_0 + h\rangle$):

$$\delta_A(j) = d_A(i_0 - h, j) - d_A(i_0, j) \qquad \delta_B(j) = d_B(j, k_0 + h) - d_B(j, k_0)$$

Sequence δ_A is monotonically increasing from $\delta_A(0) = 0$ to $\delta_A(n) \le h$. Sequence δ_B is monotonically decreasing from $\delta_B(0) \le h$ to $\delta_B(n) = 0$.

As the block size h gets smaller, sequences δ_A, δ_B contain fewer and fewer distinct values. We represent these sequences compactly by storing, for every $d \in [-\delta_B(0) : \delta_A(n)]$, the values

$$\Delta_A(d) = \operatorname*{any}_{j:\delta_A(j)-\delta_B(j)=d} \delta_A(j) \qquad \Delta_B(d) = \operatorname*{any}_{j:\delta_A(j)-\delta_B(j)=d} \delta_B(j)$$
$$M(d) = \min_{j:\delta_A(j)-\delta_B(j)=d} [d_A(i_0, j) + d_B(j, k_0)]$$

When the set $\{j : \delta_A(j) - \delta_B(j) = d\}$ is empty, the corresponding values $\Delta_A(d)$, $\Delta_B(d)$, $M(d)$ are undefined and omitted from further computations. Sequences Δ_A, Δ_B can be computed in time $O(h)$ by a single scan of the set of relevant A- and B-critical points. Sequence M is computed in the previous recursive step by a procedure described below. From sequences Δ_A, Δ_B, M, the following values can be found in time $O(h)$:

$$d_C(i_0, k_0) = \min_{d\in[-\delta_B(0):\delta_A(n)]} M(d)$$
$$d_C(i_0 - h, k_0) = \min_{d\in[-\delta_B(0):\delta_A(n)]} [\Delta_A(d) + M(d)]$$
$$d_C(i_0, k_0 + h) = \min_{d\in[-\delta_B(0):\delta_A(n)]} [M(d) + \Delta_B(d)]$$
$$d_C(i_0 - h, k_0 + h) = \min_{d\in[-\delta_B(0):\delta_A(n)]} [\Delta_A(d) + M(d) + \Delta_B(d)]$$

The number of critical points in the current block can then be determined as

$$d_C(i_0 - h, k_0 + h) - d_C(i_0 - h, k_0) - d_C(i_0, k_0 + h) + d_C(i_0, k_0)$$

If the above value is non-zero, the recursion proceeds by partitioning the current block of size h into four subblocks of size $h/2$. The sets of relevant A- and B-critical points are split accordingly. Consider each of the four half-sized subblocks. Let i'_0, k'_0, δ'_A, δ'_B, M' denote the values defined for the subblock analogously to values i_0, k_0, δ_A, δ_B, M for the original block. For every $d \in [-\delta_B(0) : \delta_A(n)]$, let

$$\Delta^*_A(d) = \operatorname*{any}_{j:\delta_A(j)-\delta_B(j)=d} \delta'_A(j) \qquad \Delta^*_B(d) = \operatorname*{any}_{j:\delta_A(j)-\delta_B(j)=d} \delta'_B(j)$$

Similarly to Δ_A, Δ_B, sequences Δ_A^*, Δ_B^* can be computed in time $O(h)$ by a single scan of the set of relevant A- and B-critical points. For every $d' \in [-\delta'_B(0) : \delta'_A(n)]$, value $M'(d')$ can now be obtained from sequence M by

$$M'(d') = \min_{d:\Delta_A^*(d)-\Delta_B^*(d)=d'} M(d) \quad \text{for } i'_0 = i_0,\ k'_0 = k_0$$

$$M'(d') = \min_{d:\Delta_A^*(d)-\Delta_B^*(d)=d'} [\Delta_A^*(d) + M(d)] \quad \text{for } i'_0 = i_0 - \tfrac{h}{2},\ k'_0 = k_0$$

$$M'(d') = \min_{d:\Delta_A^*(d)-\Delta_B^*(d)=d'} [M(d) + \Delta_B^*(d)] \quad \text{for } i'_0 = i_0,\ k'_0 = k_0 + \tfrac{h}{2}$$

$$M'(d') = \min_{d:\Delta_A^*(d)-\Delta_B^*(d)=d'} [\Delta_A^*(d) + M(d) + \Delta_B^*(d)] \quad \text{for } i'_0 = i_0 - \tfrac{h}{2},\ k'_0 = k_0 + \tfrac{h}{2}$$

Note that evaluation of functions d_A, d_B is not required. For each of the four subblocks, every value $M(d)$ contributes to exactly one value $M'(d')$, therefore the above computation can be done in time $O(h)$.

The recursion base is $h = 1$. At this point, we establish all 1×1 blocks containing a C-critical point, which is equivalent to establishing the C-critical points themselves.

Cost analysis. *First stage.* The first stage runs in a single superstep, each processor performs $O(mn/p)$ local computation and requires $O(n)$ memory.

Second stage. The second stage runs in $\log p$ supersteps. In every superstep, we have a recursion tree of maximum degree 4, height at most $\log n$, and at most n leaves (corresponding to C-critical points).

Consider the top $\frac{\log n}{2}$ levels of the recursion tree. As we move down from the root to level $\frac{\log n}{2}$, in each level the maximum number of nodes increases by a factor of 4, and the maximum amount of computational work per node decreases by a factor of 2. Hence, the maximum amount of work per level increases in geometric progression, and is dominated by level $\frac{\log n}{2}$.

Consider the bottom $\frac{\log n}{2}$ levels of the recursion tree. Since the tree has at most n leaves, the maximum number of nodes in a level is at most n. As we move down from level $\frac{\log n}{2}$ to level $\log n$, in each level the maximum amount of computational work per node still decreases by a factor of 2. Hence, the maximum amount of work per level decreases in geometric progression, and is again dominated by level $\frac{\log n}{2}$.

Thus, the computational work in the whole recursion tree is dominated by the maximum amount of work done in level $\frac{\log n}{2}$. This level has at most n nodes, each requiring at most $O(n)/2^{\frac{\log n}{2}} = O(n^{1/2})$ work. Therefore, the local computation cost of merging two strips is at most $n \cdot O(n^{1/2}) = O(n^{3/2})$, and the overall local computation cost of the second stage is $O(n^{3/2} \log p)$.

In every superstep, the recursion tree can be evaluated depth-first. Therefore, at every given moment we are only required to store the data of the current node and its ancestors in the recursion tree. As we move down from the root to the current node, in each level the maximum amount of memory required per node decreases by a factor of 2. Hence, the overall memory required for merging two strips is dominated by the $O(n)$ memory required by the root. Therefore, the overall memory cost of the second stage is $O(n)$.

The amount of data communicated between successive supersteps is $O(n)$, therefore the communication cost of the second stage is $O(n \log p)$.

Due to the slackness assumption $m \geq n^{1/2} p \log p$, the local computation cost of the algorithm is dominated by the first stage. Overall, the algorithm runs in $O(mn/p)$ local computation, $O(n \log p)$ communication, $O(\log p)$ synchronisation, and requires $O(n)$ memory. ■

Alternatively to the second stage above, the strip representations from individual processors can be collected in a single designated processor and merged sequentially. This version of the algorithm has communication cost $O(np)$ (which is higher than that of Algorithm 1, but still an improvement over [2]), and optimal synchronisation cost $O(1)$.

As before, application of Theorems 3, 4 can significantly improve the query efficiency of our algorithm's output.

5. Conclusions

We have presented a new approach to the computation of string-substring longest common subsequences. Our approach results in a significantly improved output representation, and a coarse-grained parallel algorithm for the SS-LCS problem with improved communication and memory costs.

An immediate open question is whether the efficiency of our parallel algorithm can be improved even further, with the ultimate goal of the optimal $O\left(\frac{m+n}{p}\right)$ communication and memory, and optimal $O(1)$ synchronisation (although a solution that could achieve both of these simultaneously seems unlikely). These goals are not currently achieved even for the standard LCS problem (also known as the Levenshtein distance problem). It would also be desirable to extend the algorithm to lower values of m relative to n and p. Another interesting question is whether our algorithms can be adapted to more general sequence alignment, e.g. the general edit distance problem, or sequence alignment with non-linear gap penalties.

References

[1] C. E. R. Alves, E. N. Cáceres, F. Dehne, and S. W. Song. Parallel dynamic programming for solving the string editing problem on a CGM/BSP. In *Proceedings of the 14th ACM SPAA*, pages 275–281, 2002.

[2] C. E. R. Alves, E. N. Cáceres, and S. W. Song. A BSP/CGM algorithm for the all-substrings longest common subsequence problem. In *Proceedings of the 17th IEEE/ACM IPDPS*, pages 1–8, 2003.

[3] C. E. R. Alves, E. N. Cáceres, and S. W. Song. An all-substrings common subsequence algorithm. *Electronic Notes in Discrete Mathematics*, 19:133–139, 2005.

[4] J. L. Bentley. Multidimensional divide-and-conquer. *Communications of the ACM*, 23(4):214–229, 1980.

[5] R. E. Burkard, B. Klinz, and R. Rudolf. Perspectives of Monge properties in optimization. *Discrete Applied Mathematics*, 70:95–161, 1996.

[6] J. JaJa, C. Mortensen, and Q. Shi. Space-efficient and fast algorithms for multidimensional dominance reporting and counting. In R. Fleischer and G. Trippen, editors, *Proceedings of the 15th ISAAC*, volume 3341 of *Lecture Notes in Computer Science*, pages 558–568, 2004.

[7] N. C. Jones and P. A. Pevzner. *An introduction to bioinformatics algorithms*. Computational Molecular Biology. The MIT Press, 2004.

[8] Mi Lu and Hua Lin. Parallel algorithms for the longest common subsequence problem. *IEEE Transactions on Parallel and Distributed Systems*, 5(8):835–848, 1994.

[9] J. P. Schmidt. All highest scoring paths in weighted grid graphs and their application to finding all approximate repeats in strings. *SIAM Journal on Computing*, 27(4):972–992, 1998.

[10] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.